# CHAPTER 5: CONTROLLING WEB APPLICATION BEHAVIOR WITH WEB.XML

Update from Marty: For examples from this chapter that have been updated to the latest specification, please see http://courses.coreservlets.com/Course-Materials/msajsp.html.

## Topics in This Chapter

- Customizing URLs
- Turning off default URLs
- Initializing servlets and JSP pages
- Preloading servlets and JSP pages
- Declaring filters for servlets and JSP pages
- Designating welcome pages and error pages
- Restricting access to Web resources
- Controlling session timeouts
- Documenting Web applications
- Specifying MIME types
- Locating tag library descriptors
- Declaring event listeners
- Accessing J2EE resources

# Chapter 5

This chapter describes the makeup of the deployment descriptor file that is placed in *WEB-INF/web.xml* within each Web application. I'll summarize all the legal elements here; for the formal specification see *http://java.sun.com/dtd/web-app_2_3.dtd* (for version 2.3 of the servlet API) or *http://java.sun.com/j2ee/dtds/web-app_2_2.dtd* (for version 2.2).

Most of the servlet and JSP examples in this chapter assume that they are part of a Web application named `deployDemo`. For details on how to set up and register Web applications, please see Chapter 4 (Chapter 4: Using and Deploying Web Applications).

# 5.1 Defining the Header and Root Elements

The deployment descriptor, like all XML files, must begin with an XML header. This header declares the version of XML that is in effect and gives the character encoding for the file.

A `DOCTYPE` declaration must appear immediately after the header. This declaration tells the server the version of the servlet specification (e.g., 2.2 or 2.3) that applies and specifies the Document Type Definition (DTD) that governs the syntax of the rest of the file.

The top-level (root) element for all deployment descriptors is `web-app`. Remember that XML elements, unlike HTML elements, are case sensitive. Consequently, `Web-App` and `WEB-APP` are not legal; you must use `web-app` in lower case.

### Core Warning

*XML elements are case sensitive.*

Thus, the *web.xml* file should be structured as follows for Web apps that will run in servlet 2.2 containers (servers) or that will run in 2.3 containers but not make use of the new *web.xml* capabilities (e.g., filter or listener declarations) introduced in version 2.3.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <!-- Other elements go here. All are optional. -->
</web-app>
```

Rather than typing in this template by hand, you can (and should) download an example from *http://www.moreservlets.com* or copy and edit the version that comes in the default Web application of whatever server you use. However, note that the *web.xml* file that is distributed with Allaire JRun 3 (e.g., in *install_dir/servers/default/default-app/WEB-INF* for JRun 3.0 or the *samples* directory for JRun 3.1) incorrectly omits the XML header and `DOCTYPE` line. As a result, although JRun accepts properly formatted *web.xml* files from other servers, other servers might not accept the *web.xml* file from JRun 3. So, if you use JRun 3, be sure to insert the header and `DOCTYPE` lines.

### Core Warning

*The* web.xml *file that is distributed with JRun 3 is illegal; it is missing the XML header and* DOCTYPE *declaration.*

If you want to use servlet/JSP filters, application life-cycle listeners, or other features specific to servlets 2.3, you must use the servlet 2.3 DTD, as shown in the *web.xml* file below. Of course, you must also use a server that supports this version of the specification—version 4 of Tomcat, JRun, or ServletExec, for example. Just be

aware that your Web application will not run in servers that support only version 2.2 of the servlet API (e.g., version 3 of Tomcat, JRun, or ServletExec).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <!-- Other elements go here. All are optional. -->
</web-app>
```

# 5.2 The Order of Elements within the Deployment Descriptor

Not only are XML elements case sensitive, they are also sensitive to the order in which they appear within other elements. For example, the XML header must be the first entry in the file, the DOCTYPE declaration must be second, and the web-app element must be third. Within the web-app element, the order of the elements also matters. Servers are not required to enforce this ordering, but they are permitted to, and some do so in practice, completely refusing to run Web applications that contain elements that are out of order. This means that *web.xml* files that use nonstandard element ordering are not portable.

### Core Approach

*Be sure to correctly order the elements that appear within web-app.*

The following list gives the required ordering of all legal elements that can appear directly within the web-app element. For example, the list shows that any servlet elements must appear before any servlet-mapping elements. If there are any mime-mapping elements, they must go after all servlet and servlet-mapping elements but before welcome-file-list. Remember that all these elements are optional. So, you can omit any element but you cannot place it in a nonstandard location.

- **icon**. The icon element designates the location of either one or two image files that an IDE or GUI tool can use to represent the Web application. For details, see Section 5.11 (Documenting Web Applications).

- **display-name**. The `display-name` element provides a name that GUI tools might use to label this particular Web application. See Section 5.11.
- **description**. The `description` element gives explanatory text about the Web application. See Section 5.11.

- **distributable**. The `distributable` element tells the system that it is safe to distribute the Web application across multiple servers. See Section 5.15.
- **context-param**. The `context-param` element declares application-wide initialization parameters. For details, see Section 5.5 (Initializing and Preloading Servlets and JSP Pages).
- **filter**. The filter element associates a name with a class that implements the `javax.servlet.Filter` interface. For details, see Section 5.6 (Declaring Filters).
- **filter-mapping**. Once you have named a filter, you associate it with one or more servlets or JSP pages by means of the `filter-mapping` element. See Section 5.6.
- **listener**. Version 2.3 of the servlet API added support for event listeners that are notified when the session or servlet context is created, modified, or destroyed. The `listener` element designates the event listener class. See Section 5.14 for details.
- **servlet**. Before you assign initialization parameters or custom URLs to servlets or JSP pages, you must first name the servlet or JSP page. You use the `servlet` element for that purpose. For details, see Section 5.3.
- **servlet-mapping**. Servers typically provide a default URL for servlets: *http://host/webAppPrefix/servlet/ServletName*. However, you often change this URL so that the servlet can access initialization parameters (Section 5.5) or more easily handle relative URLs (Section 4.5). When you do change the default URL, you use the `servlet-mapping` element to do so. See Section 5.3.
- **session-config**. If a session has not been accessed for a certain period of time, the server can throw it away to save memory. You can explicitly set the timeout for individual session objects by using the `setMaxInactiveInterval` method of `HttpSession`, or you can use the `session-config` element to designate a default timeout. For details, see Section 5.10.
- **mime-mapping**. If your Web application has unusual files that you want to guarantee are assigned certain MIME types, the `mime-mapping` element can provide this guarantee. For more information, see Section 5.12.

- **welcome-file-list**. The welcome-file-list element instructs the server what file to use when the server receives URLs that refer to a directory name but not a filename. See Section 5.7 for details.

- **error-page**. The error-page element lets you designate the pages that will be displayed when certain HTTP status codes are returned or when certain types of exceptions are thrown. For details, see Section 5.8.

- **taglib**. The taglib element assigns aliases to Tag Library Descriptor files. This capability lets you change the location of the TLD files without editing the JSP pages that use those files. See Section 5.13 for more information.

- **resource-env-ref**. The resource-env-ref element declares an administered object associated with a resource. See Section 5.15.

- **resource-ref**. The resource-ref element declares an external resource used with a resource factory. See Section 5.15.

- **security-constraint**. The security-constraint element lets you designate URLs that should be protected. It goes hand-in-hand with the login-config element. See Section 5.9 for details.

- **login-config**. You use the login-config element to specify how the server should authorize users who attempt to access protected pages. It goes hand-in-hand with the security-constraint element. See Section 5.9 for details.

- **security-role**. The security-role element gives a list of security roles that will appear in the role-name subelements of the security-role-ref element inside the servlet element. Declaring the roles separately could make it easier for advanced IDEs to manipulate security information. See Section 5.9 for details.

- **env-entry**. The env-entry element declares the Web application's environment entry. See Section 5.15.

- **ejb-ref**. The ejb-ref element declares a reference to the home of an enterprise bean. See Section 5.15.

- **ejb-local-ref**. The ejb-local-ref element declares a reference to the local home of an enterprise bean. See Section 5.15.

# 5.3 Assigning Names and Custom URLs

One of the most common tasks that you perform in *web.xml* is giving names and custom URLs to your servlets or JSP pages. You use the `servlet` element to assign names; you use the `servlet-mapping` element to associate custom URLs with the name just assigned.

## Assigning Names

In order to provide initialization parameters, define a custom URL, or assign a security role to a servlet or JSP page, you must first give the servlet or page a name. You assign a name by means of the `servlet` element. The most common format includes `servlet-name` and `servlet-class` subelements (inside the `web-app` element), as follows.

```
<servlet>

  <servlet-name>Test</servlet-name>

  <servlet-class>moreservlets.TestServlet</servlet-class>

</servlet>
```

This means that the servlet at *WEB-INF/classes/moreservlets/TestServlet* is now known by the registered name `Test`. Giving a servlet a name has two major implications. First, initialization parameters, custom URL patterns, and other customizations refer to the servlet by the registered name, not by the class name. Second, the name can be used in the URL instead of the class name. Thus, with the definition just given, the URL *http://host/webAppPrefix/servlet/**Test*** can be used in lieu of *http://host/webAppPrefix/servlet/**moreservlets.TestServlet***.

Remember: not only are XML elements case sensitive, but the order in which you define them also matters. For example, all `servlet` elements within the `web-app` element must come before any of the `servlet-mapping` elements that are introduced in the next subsection, but before the filter or documentation-related elements (if any) that are discussed in Sections 5.6 and 5.11. Similarly, the `servlet-name` subelement of `servlet` must come before `servlet-class`. Section 5.2 (The Order of Elements within the Deployment Descriptor) describes the required ordering in detail.

### Core Approach

*Be sure to properly order the elements within the `web-app` element of* web.xml. *In particular, the `servlet` element must come before `servlet-mapping`.*

For example, Listing 5.1 shows a simple servlet called `TestServlet` that resides in the `moreservlets` package. Since the servlet is part of a Web application rooted in a directory named *deployDemo*, *TestServlet.class* is placed in *deployDemo/WEB-INF/classes/moreservlets*. Listing 5.2 shows a portion of the *web.xml* file that would be placed in *deployDemo/WEB-INF/*. This *web.xml* file uses the `servlet-name` and `servlet-class` elements to associate the name `Test` with *TestServlet.class*. Figures 5–1 and 5–2 show the results when `TestServlet` is invoked by means of the default URL and the registered name, respectively.

---

**Listing 5.1**  *TestServlet.java*

```
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Simple servlet used to illustrate servlet naming
 *  and custom URLs.
 */

public class TestServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String uri = request.getRequestURI();
    out.println(ServletUtilities.headWithTitle("Test Servlet") +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H2>URI: " + uri + "</H2>\n" +
                "</BODY></HTML>");
  }
}
```

---

| **Listing 5.2** | *web.xml* (Excerpt showing servlet name) |
|---|---|

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->
  <servlet>
    <servlet-name>Test</servlet-name>
    <servlet-class>moreservlets.TestServlet</servlet-class>
  </servlet>
  <!-- ... -->
</web-app>
```
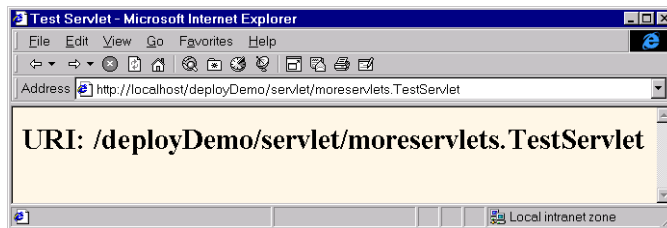


**Figure 5–1**  `TestServlet` when invoked with the default URL.
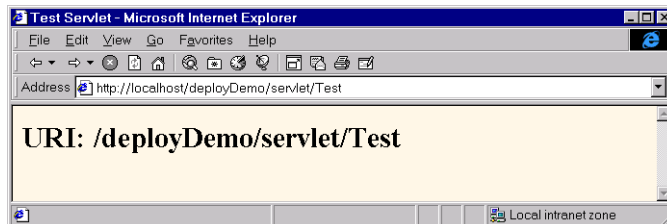


**Figure 5–2**  `TestServlet` when invoked with the registered name.

## Defining Custom URLs

Most servers have a default URL for servlets: *http://host/webAppPrefix/servlet/packageName.ServletName*. Although it is convenient to use this URL during development, you often want a different URL for deployment. For instance, you might want a URL that appears in the top level of the Web application (e.g., *http://host/*

*webAppPrefix/AnyName*), without the *servlet* entry in the URL. A URL at the top level simplifies the use of relative URLs, as described in Section 4.5 (Handling Relative URLs in Web Applications). Besides, top-level URLs are shorter and simply *look* better to many developers than the long and cumbersome default URLs.

In fact, sometimes you are *required* to use a custom URL. For example, you might turn off the default URL mapping so as to better enforce security restrictions or to prevent users from accidentally accessing servlets that have no init parameters. This idea is discussed further in Section 5.4 (Disabling the Invoker Servlet). If you disable the default URL, how do you access the servlet? Only by using a custom URL.

To assign a custom URL, you use the `servlet-mapping` element along with its `servlet-name` and `url-pattern` subelements. The `servlet-name` element provides an arbitrary name with which to refer to the servlet; `url-pattern` describes a URL relative to the Web application root. The value of the `url-pattern` element must begin with a slash (`/`).

Here is a simple *web.xml* excerpt that lets you use the URL *http://host/webApp-Prefix/UrlTest* instead of either *http://host/webAppPrefix/servlet/Test* or *http://host/webAppPrefix/servlet/moreservlets.TestServlet*. Figure 5–3 shows a typical result. Remember that you still need the XML header, the `DOCTYPE` declaration, and the enclosing `web-app` element as described in Section 5.1 (Defining the Header and Root Elements). Furthermore, recall that the order in which XML elements appears is not arbitrary. In particular, you are required to put all the `servlet` elements before any of the `servlet-mapping` elements. For a complete breakdown of the required ordering of elements within `web-app`, see Section 5.2 (The Order of Elements within the Deployment Descriptor).

```
<servlet>

  <servlet-name>Test</servlet-name>

  <servlet-class>moreservlets.TestServlet</servlet-class>

</servlet>

<!-- ... -->

<servlet-mapping>

  <servlet-name>Test</servlet-name>

  <url-pattern>/UrlTest</url-pattern>

</servlet-mapping>
```
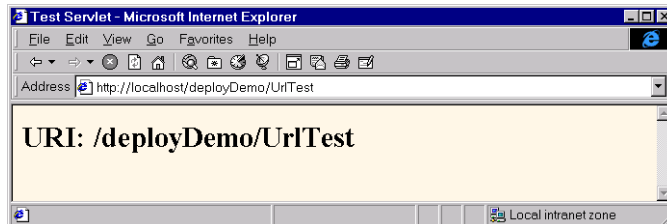
**Figure 5–3**    `TestServlet` invoked with a URL pattern.

The URL pattern can also include wildcards. For example, the following snippet instructs the server to send to all requests beginning with the Web app's URL prefix (see Section 4.1, "Registering Web Applications") and ending with *.asp* to the servlet named `BashMS`.

```
<servlet>
  <servlet-name>BashMS</servlet-name>
  <servlet-class>msUtils.ASPTranslator</servlet-class>
</servlet>
<!-- ... -->
<servlet-mapping>
  <servlet-name>BashMS</servlet-name>
  <url-pattern>/*.asp</url-pattern>
</servlet-mapping>
```

## Naming JSP Pages

Since JSP pages get translated into servlets, it is natural to expect that you can name JSP pages just as you can name servlets. After all, JSP pages might benefit from initialization parameters, security settings, or custom URLs, just as regular servlets do. Although it is true that JSP pages are really servlets behind the scenes, there is one key difference: you don't know the actual class name of JSP pages (since the system picks the name). So, to name JSP pages, you substitute the `jsp-file` element for the `servlet-class` element, as follows.

```
<servlet>
  <servlet-name>PageName</servlet-name>
  <jsp-file>/TestPage.jsp</jsp-file>
</servlet>
```

You name JSP pages for exactly the same reason that you name servlets: to provide a name to use with customization settings (e.g., initialization parameters and security

settings) and so that you can change the URL that invokes the JSP page (e.g., so that multiple URLs get handled by the same page or to remove the *.jsp* extension from the URL). However, when setting initialization parameters, remember that JSP pages read initialization parameters by using the `jspInit` method, not the `init` method. See Section 5.5 (Initializing and Preloading Servlets and JSP Pages) for details.

For example, Listing 5.3 is a simple JSP page named *TestPage.jsp* that just prints out the local part of the URL used to invoke it. *TestPage.jsp* is placed in the top level of the *deployDemo* directory. Listing 5.4 shows a portion of the *web.xml* file (i.e., *deployDemo/WEB-INF/web.xml*) used to assign a registered name of `PageName` and then to associate that registered name with URLs of the form *http://host/webApp-Prefix/UrlTest2/anything*. Figures 5–4 through 5–6 show the results for the URLs *http://localhost/deployDemo/TestPage.jsp* (the real name), *http://localhost/deploy-Demo/servlet/PageName* (the registered servlet name), and *http://localhost/deploy-Demo/UrlTest2/foo/bar/baz.html* (a URL matching `url-pattern`), respectively.

---

**Listing 5.3**    *TestPage.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>JSP Test Page</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2>URI:  <%= request.getRequestURI() %></H2>
</BODY></HTML>
```

---

| **Listing 5.4** | *web.xml* (Excerpt illustrating the naming of JSP pages) |
|---|---|

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->
  <servlet>
    <servlet-name>PageName</servlet-name>
    <jsp-file>/TestPage.jsp</jsp-file>
  </servlet>
  <!-- ... -->
  <servlet-mapping>
    <servlet-name>PageName</servlet-name>
    <url-pattern>/UrlTest2/*</url-pattern>
  </servlet-mapping>
  <!-- ... -->
</web-app>
```

---
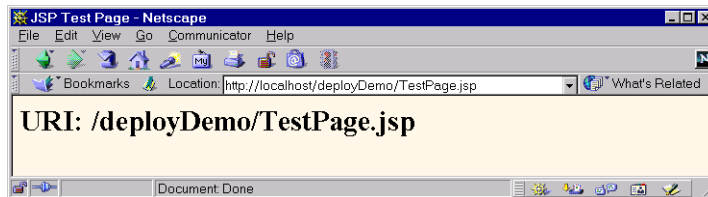


**Figure 5–4**    *TestPage.jsp* invoked with the normal URL.
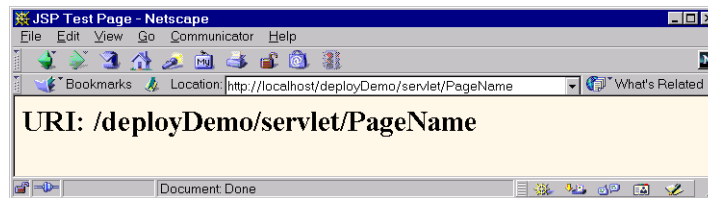


**Figure 5–5**    *TestPage.jsp* invoked with the registered servlet name.

**Figure 5–6** *TestPage.jsp* invoked with a URL that matches the designated
`url-pattern`.

# 5.4 Disabling the Invoker Servlet

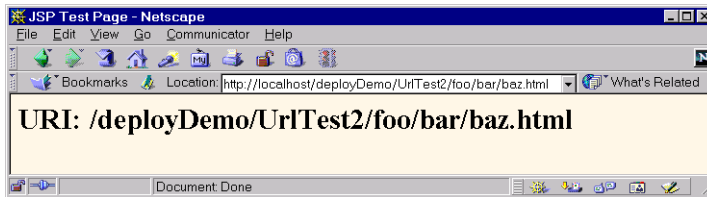One reason for setting up a custom URL for a servlet or JSP page is so that you can register initialization parameters to be read from the `init` (servlets) or `jspInit` (JSP pages) methods. However, as discussed in Section 5.5 (Initializing and Preloading Servlets and JSP Pages), the initialization parameters are available only when the servlet or JSP page is accessed by means of a custom URL pattern or a registered name, not when it is accessed with the default URL of *http://host/webAppPrefix/ servlet/ServletName*. Consequently, you might want to turn off the default URL so that nobody accidentally calls the uninitialized servlet. This process is sometimes known as disabling *the invoker servlet*, since most servers have a standard servlet that is registered with the default servlet URLs and simply invokes the real servlet that the URL refers to.

There are two main approaches for disabling the default URL:

- Remapping the */servlet/* pattern in each Web application.
- Globally turning off the invoker servlet.

It is important to note that, although remapping the */servlet/* pattern in each Web application is more work than disabling the invoker servlet in one fell swoop, remapping can be done in a completely portable manner. In contrast, the process for globally disabling the invoker servlet is completely machine specific, and in fact some servers (e.g., ServletExec) have no such option. The first following subsection discusses the per-Web-application strategy of remapping the */servlet/* URL pattern. The next two subsections provide details on globally disabling the invoker servlet in Tomcat and JRun.

## Remapping the /servlet/ URL Pattern

It is quite straightforward to disable processing of URLs that begin with *http://host/ webAppPrefix/servlet/* in a particular Web application. All you need to do is create an

error message servlet and use the `url-pattern` element discussed in the previous section to direct all matching requests to that servlet. Simply use

```
<url-pattern>/servlet/*</url-pattern>
```

as the pattern within the `servlet-mapping` element.

For example, Listing 5.5 shows a portion of the deployment descriptor that associates the `SorryServlet` servlet (Listing 5.6) with all URLs that begin with *http://host/ webAppPrefix/servlet/*. Figures 5–7 and 5–8 illustrate attempts to access the `Test-Servlet` servlet (Listing 5.1 in Section 5.3) before (Figure 5–7) and after (Figure 5–8) the *web.xml* entries of Listing 5.5 are made.

All compliant servers yield the results of Figures 5–7 and 5–8. However, Servlet-Exec 4.0 has a bug whereby mappings of the `/servlet/*` pattern are ignored (other mappings work fine). Furthermore, since ServletExec has no global method of disabling the invoker servlet, in version 4.0 you are left with no alternative but to leave the invoker servlet enabled. The problem is resolved in ServletExec version 4.1.

### Core Warning

*You cannot disable the invoker servlet in ServletExec 4.0.*

---

| **Listing 5.5** | *web.xml* (Excerpt showing how to disable default URLs) |
|---|---|

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->
  <servlet>
    <servlet-name>Sorry</servlet-name>
    <servlet-class>moreservlets.SorryServlet</servlet-class>
  </servlet>
  <!-- ... -->
  <servlet-mapping>
    <servlet-name>Sorry</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
  <!-- ... -->
</web-app>
```

**Listing 5.6** *SorryServlet.java*

```java
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Simple servlet used to give error messages to
 *  users who try to access default servlet URLs
 *  (i.e., http://host/webAppPrefix/servlet/ServletName)
 *  in Web applications that have disabled this
 *  behavior.
 */

public class SorryServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Invoker Servlet Disabled.";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H2>" + title + "</H2>\n" +
                "Sorry, access to servlets by means of\n" +
                "URLs that begin with\n" +
                "http://host/webAppPrefix/servlet/\n" +
                "has been disabled.\n" +
                "</BODY></HTML>");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```
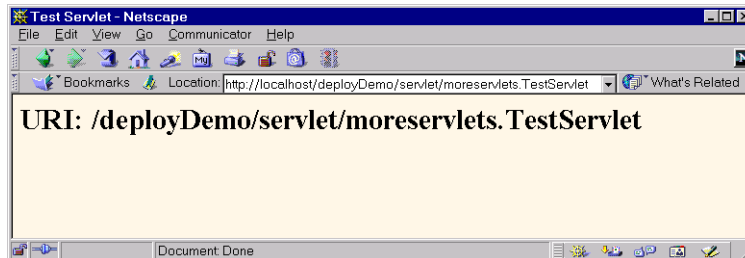
**Figure 5–7**    Successful attempt to invoke the `TestServlet` servlet by means of the default URL. The invoker servlet is enabled.
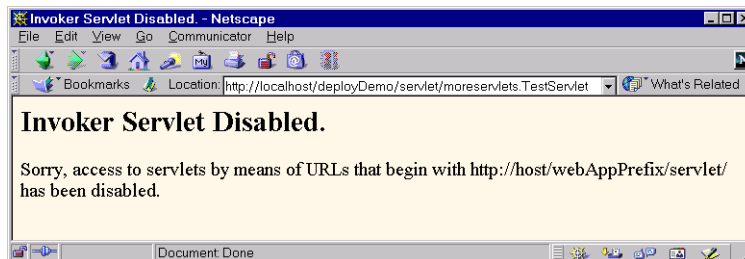


**Figure 5–8**    Unsuccessful attempt to invoke the `TestServlet` servlet by means of the default URL. The invoker servlet is disabled.

# Globally Disabling the Invoker: Tomcat

The method you use to turn off the default URL in Tomcat 4 is quite different from the approach used in Tomcat 3. The following two subsections summarize the two approaches.

## Disabling the Invoker: Tomcat 4

Tomcat 4 turns off the invoker servlet in the same way that I turned it off in the previous section: by means of a `url-mapping` element in *web.xml*. The difference is that Tomcat uses a server-specific global *web.xml* file that is stored in *install_dir/conf*, whereas I used the standard *web.xml* file that is stored in the *WEB-INF* directory of each Web application.

Thus, to turn off the invoker servlet in Tomcat 4, you simply comment out the `/servlet/*` URL mapping entry in *install_dir/conf/web.xml*, as shown below.

```
<!--
<servlet-mapping>
  <servlet-name>invoker</servlet-name>
```

```
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
-->
```

Again, note that this entry is in the Tomcat-specific *web.xml* file that is stored in *install_dir/conf*, not the standard *web.xml* file that is stored in the *WEB-INF* directory of each Web application.

Figures 5–9 and 5–10 show the results when the `TestServlet` (Listing 5.1 from the previous section) is invoked with the default URL and with the registered servlet name in a version of Tomcat that has the invoker servlet disabled. Both URLs are of the form *http://host/webAppPrefix/servlet/something*, and both fail. Figure 5–11 shows the result when the explicit URL pattern is used; this request succeeds.
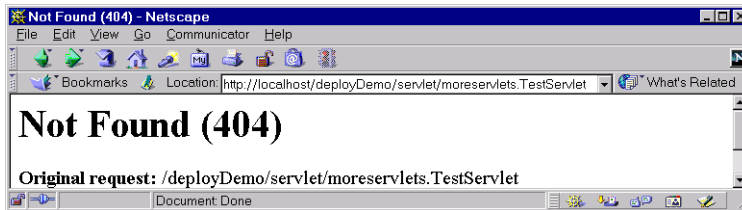


**Figure 5–9**     `TestServlet` when invoked with the default URL in a server that has globally disabled the invoker servlet.
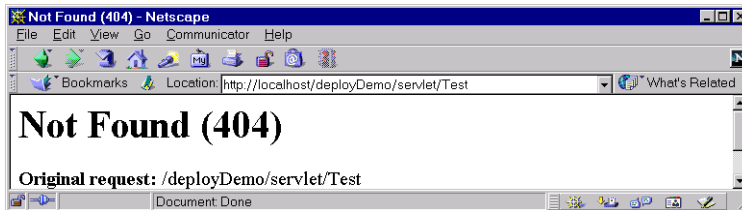


**Figure 5–10**   `TestServlet` when invoked with a registered name in a server that has globally disabled the invoker servlet.
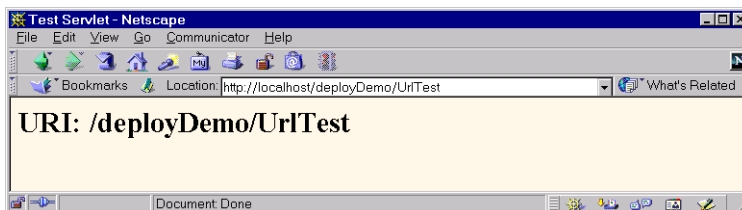


**Figure 5–11**   `TestServlet` when invoked with a custom URL in a server that has globally disabled the invoker servlet.

## Disabling the Invoker: Tomcat 3

In version 3 of Apache Tomcat, you globally disable the default servlet URL by commenting out the `InvokerInterceptor` entry in *install_dir/conf/server.xml*. For example, following is a section of a *server.xml* file that prohibits use of the default servlet URL.

```
<!--
<RequestInterceptor
  className="org.apache.tomcat.request.InvokerInterceptor"
  debug="0" prefix="/servlet/" />
-->
```

With this entry commented out, Tomcat 3 gives the same results as shown in Figures 5–9 through 5–11.

# Globally Disabling the Invoker: JRun

In JRun 3.1, you disable the invoker servlet by editing *install_dir/lib/global.properties* and inserting a # at the beginning of the line that defines the invoker, thus commenting out the line. This is illustrated below.

```
# webapp.servlet-mapping./servlet=invoker
```

With these settings, JRun gives about the same results as shown in Figures 5–9 through 5–11; the only minor difference is that it gives 500 (Internal Server Error) messages for the first two cases instead of the 404 (Not Found) messages that Tomcat gives.

# 5.5    Initializing and Preloading Servlets and JSP Pages

This section discusses methods for controlling the startup behavior of servlets and JSP pages. In particular, it explains how you can assign initialization parameters and how you can change the point in the server life cycle at which servlets and JSP pages are loaded.

## Assigning Servlet Initialization Parameters

You provide servlets with initialization parameters by means of the `init-param` element, which has `param-name` and `param-value` subelements. For instance,

in the following example, if the `InitServlet` servlet is accessed by means of its registered name (`InitTest`), it could call `getServletConfig().getInit-Parameter("param1")` from its init method to get `"Value 1"` and `get-ServletConfig().getInitParameter("param2")` to get `"2"`.

```
<servlet>
  <servlet-name>InitTest</servlet-name>
  <servlet-class>myPackage.InitServlet</servlet-class>
  <init-param>
    <param-name>param1</param-name>
    <param-value>Value 1</param-value>
  </init-param>
  <init-param>
    <param-name>param2</param-name>
    <param-value>2</param-value>
  </init-param>
</servlet>
```

There are a few common gotchas that are worth keeping in mind when dealing with initialization parameters:

- **Return values.** The return value of `getInitParameter` is always a `String`. So, for instance, in the previous example you might use `Integer.parseInt` on `param2` to obtain an `int`.
- **Initialization in JSP.** JSP pages use `jspInit`, not `init`. JSP pages also require use of the `jsp-file` element in place of `servlet-class`, as described in Section 5.3 (Assigning Names and Custom URLs). Initializing JSP pages is discussed in the next subsection.
- **Default URLs.** Initialization parameters are only available when servlets are accessed by means of their registered names or through custom URL patterns associated with their registered names. So, in this example, the `param1` and `param2` init parameters would be available when you used the URL *http://host/webAppPrefix/servlet/InitTest*, but not when you used the URL *http://host/webAppPrefix/servlet/myPackage.InitServlet*.

### Core Warning

*Initialization parameters are not available in servlets that are accessed by their default URL.*

For example, Listing 5.7 shows a simple servlet called `InitServlet` that uses the `init` method to set the `firstName` and `emailAddress` fields. Listing 5.8 shows the *web.xml* file that assigns the name `InitTest` to the servlet. Figures 5–12 and 5–13 show the results when the servlet is accessed with the registered name (correct) and the original name (incorrect), respectively.

**Listing 5.7** *InitServlet.java*

```java
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Simple servlet used to illustrate servlet
 *  initialization parameters.
 */

public class InitServlet extends HttpServlet {
  private String firstName, emailAddress;

  public void init() {
    ServletConfig config = getServletConfig();
    firstName = config.getInitParameter("firstName");
    emailAddress = config.getInitParameter("emailAddress");
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String uri = request.getRequestURI();
    out.println(ServletUtilities.headWithTitle("Init Servlet") +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H2>Init Parameters:</H2>\n" +
                "<UL>\n" +
                "<LI>First name: " + firstName + "\n" +
                "<LI>Email address: " + emailAddress + "\n" +
                "</UL>\n" +
                "</BODY></HTML>");
  }
}
```

> **Listing 5.8**    *web.xml* (Excerpt illustrating initialization parameters)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->
  <servlet>
    <servlet-name>InitTest</servlet-name>
    <servlet-class>moreservlets.InitServlet</servlet-class>
    <init-param>
      <param-name>firstName</param-name>
      <param-value>Larry</param-value>
    </init-param>
    <init-param>
      <param-name>emailAddress</param-name>
      <param-value>ellison@microsoft.com</param-value>
    </init-param>
  </servlet>
  <!-- ... -->
</web-app>
```
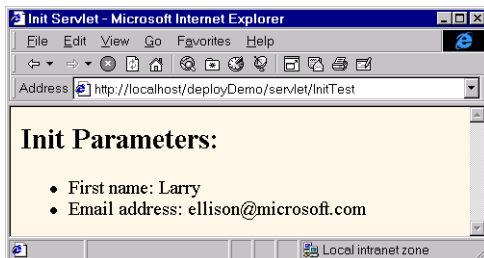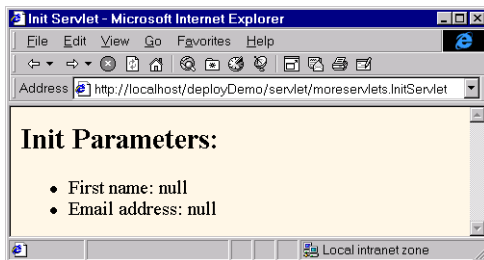


**Figure 5–12**   The `InitServlet` when correctly accessed with its registered name.



Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/

**Figure 5–13**   The `InitServlet` when incorrectly accessed with the default URL.

# Assigning JSP Initialization Parameters

Providing initialization parameters to JSP pages differs in three ways from providing them to servlets.

1. **You use `jsp-file` instead of `servlet-class`.** So, the `servlet` element of the *WEB-INF/web.xml* file would look something like this:
   ```
   <servlet>
     <servlet-name>PageName</servlet-name>
     <jsp-file>/RealPage.jsp</jsp-file>
     <init-param>
       <param-name>...</param-name>
       <param-value>...</param-value>
     </init-param>
     ...
   </servlet>
   ```

2. **You almost always assign an explicit URL pattern.** With servlets, it is moderately common to use the default URL that starts with *http://host/webAppPrefix/**servlet**/*; you just have to remember to use the registered name instead of the original name. This is technically legal with JSP pages also. For example, with the example just shown in item 1, you could use a URL of *http://host/webAppPrefix/servlet/PageName* to access the version of *RealPage.jsp* that has access to initialization parameters. But, many users dislike URLs that appear to refer to regular servlets when used for JSP pages. Furthermore, if the JSP page is in a directory for which the server provides a directory listing (e.g., a directory with neither an *index.html* nor an *index.jsp* file), the user might get a link to the JSP page, click on it, and thus accidentally invoke the uninitialized page. So, a good strategy is to use `url-pattern` (Section 5.3) to associate the *original* URL of the JSP page with the registered servlet name. That way, clients can use the normal name for the JSP page but still invoke the customized version. For example, given the `servlet` definition from item 1, you might use the following `servlet-mapping` definition:

   ```
   <servlet-mapping>
     <servlet-name>PageName</servlet-name>
     <url-pattern>/RealPage.jsp</url-pattern>
   </servlet-mapping>
   ```

3. **The JSP page uses `jspInit`, not `init`.** The servlet that is automatically built from a JSP page may already be using the `init`

method. Consequently, it is illegal to use a JSP declaration to provide an `init` method. You must name the method `jspInit` instead.

To illustrate the process of initializing JSP pages, Listing 5.9 shows a JSP page called *InitPage.jsp* that contains a `jspInit` method and is placed at the top level of the `deployDemo` Web page hierarchy. Normally, a URL of *http://localhost/deploy-Demo/InitPage.jsp* would invoke a version of the page that has no access to initialization parameters and would thus show `null` for the `firstName` and `emailAddress` variables. However, the *web.xml* file (Listing 5.10) assigns a registered name and then associates that registered name with the URL pattern */InitPage.jsp*. As Figure 5–14 shows, the result is that the normal URL for the JSP page now invokes the version of the page that has access to the initialization parameters.

---

**Listing 5.9**    *InitPage.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>JSP Init Test</TITLE></HEAD>

<BODY BGCOLOR="#FDF5E6">

<H2>Init Parameters:</H2>
<UL>
  <LI>First name: <%= firstName %>
  <LI>Email address: <%= emailAddress %>
</UL>

</BODY></HTML>

<%!
private String firstName, emailAddress;

public void jspInit() {
  ServletConfig config = getServletConfig();
  firstName = config.getInitParameter("firstName");
  emailAddress = config.getInitParameter("emailAddress");
}
%>
```

---

| **Listing 5.10** | *web.xml* (Excerpt showing init params for JSP pages) |
|---|---|

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->
  <servlet>
    <servlet-name>InitPage</servlet-name>
    <jsp-file>/InitPage.jsp</jsp-file>
    <init-param>
      <param-name>firstName</param-name>
      <param-value>Bill</param-value>
    </init-param>
    <init-param>
      <param-name>emailAddress</param-name>
      <param-value>gates@oracle.com</param-value>
    </init-param>
  </servlet>
  <!-- ... -->
  <servlet-mapping>
    <servlet-name>InitPage</servlet-name>
    <url-pattern>/InitPage.jsp</url-pattern>
  </servlet-mapping>
  <!-- ... -->
</web-app>
```
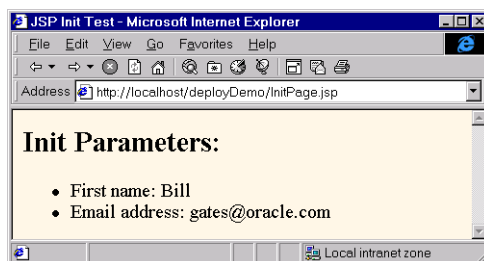


**Figure 5–14**   Mapping a JSP page's original URL to the registered servlet name prevents users from accidentally accessing the uninitialized version.

# Supplying Application-Wide Initialization Parameters

Normally, you assign initialization parameters to individual servlets or JSP pages. The designated servlet or JSP page reads the parameters by means of the `getInit-Parameter` method of Servlet**Config**. However, in some situations you want to supply system-wide initialization parameters that can be read by *any* servlet or JSP page by means of the `getInitParameter` method of Servlet**Context**.

You use the `context-param` element to declare these system-wide initialization values. The `context-param` element should contain `param-name`, `param-value`, and, optionally, `description` subelements, as below.

```
<context-param>
  <param-name>support-email</param-name>
  <param-value>blackhole@mycompany.com</param-value>
</context-param>
```

Recall that, to ensure portability, the elements within *web.xml* must be declared in the proper order. Complete details are given in Section 5.2 (The Order of Elements within the Deployment Descriptor). Here, however, just note that the `context-param` element must appear after any documentation-related elements (`icon`, `display-name`, and `description`—see Section 5.11) and before any `filter` (Section 5.6), `filter-mapping` (Section 5.6), `listener` (Section 5.14), or `servlet` (Section 5.3) elements.

# Loading Servlets When the Server Starts

Suppose that a servlet or JSP page has an `init` (servlet) or `jspInit` (JSP) method that takes a long time to execute. For example, suppose that the `init` or `jspInit` method looks up constants from a database or `ResourceBundle`. In such a case, the default behavior of loading the servlet at the time of the first client request results in a significant delay for that first client. So, you can use the `load-on-startup` subelement of `servlet` to stipulate that the server load the servlet when the server first starts. Here is an example.

```
<servlet>
  <servlet-name>...</servlet-name>
  <servlet-class>...</servlet-class> <!-- Or jsp-file -->
  <load-on-startup />
</servlet>
```

Rather than using an empty `load-on-startup` element, you can supply an integer for the element body. The idea is that the server should load lower-numbered servlets or JSP pages before higher-numbered ones. For example, the following

`servlet` entries (placed within the `web-app` element in the *web.xml* file that goes in the *WEB-INF* directory of your Web application) would instruct the server to first load and initialize `SearchServlet`, then load and initialize the servlet resulting from the *index.jsp* file that is in the Web app's *results* directory.

```
<servlet>
  <servlet-name>Search</servlet-name>
  <servlet-class>myPackage.SearchServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet>
  <servlet-name>Results</servlet-name>
  <jsp-file>/results/index.jsp</jsp-file>
  <load-on-startup>2</load-on-startup>
</servlet>
```

# 5.6 Declaring Filters

Servlet version 2.3 introduced the concept of filters. Although filters are supported by all servers that support version 2.3 of the servlet API, you must use the version 2.3 DTD in *web.xml* in order to use the filter-related elements. For details on the DTD, see Section 5.1 (Defining the Header and Root Elements).

**Core Note**

> *Filters and the filter-related elements in* web.xml *are available only in servers that support the Java servlet API version 2.3. Even with compliant servers, you must use the version 2.3 DTD.*

Filters are discussed in detail in Chapter 9, but the basic idea is that filters can intercept and modify the request coming into or the response going out of a servlet or JSP page. Before a servlet or JSP page is executed, the `doFilter` method of the first associated filter is executed. When that filter calls `doFilter` on its `Filter-Chain` object, the next filter in the chain is executed. If there is no other filter, the servlet or JSP page itself is executed. Filters have full access to the incoming `ServletRequest` object, so they can check the client's hostname, look for incoming cookies, and so forth. To access the output of the servlet or JSP page, a filter can wrap the response object inside a stand-in object that, for example, accumulates the output into a buffer. After the call to the `doFilter` method of the `FilterChain`

object, the filter can examine the buffer, modify it if necessary, and then pass it on to the client.

For example, Listing 5.11 defines a simple filter that intercepts requests and prints a report on the standard output (available with most servers when you run them on your desktop during development) whenever the associated servlet or JSP page is accessed.

---

**Listing 5.11**   *ReportFilter.java*

```java
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Simple filter that prints a report on the standard output
 *  whenever the associated servlet or JSP page is accessed.
 */

public class ReportFilter implements Filter {
  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)
      throws ServletException, IOException {
    HttpServletRequest req = (HttpServletRequest)request;
    System.out.println(req.getRemoteHost() +
                       " tried to access " +
                       req.getRequestURL() +
                       " on " + new Date() + ".");
    chain.doFilter(request,response);
  }

  public void init(FilterConfig config)
      throws ServletException {
  }

  public void destroy() {}
}
```

---

Once you have created a filter, you declare it in the *web.xml* file by using the `filter` element along with the `filter-name` (arbitrary name), `filter-class` (fully qualified class name), and, optionally, `init-params` subelements. Remember that the order in which elements appear within the `web-app` element of *web.xml* is

not arbitrary; servers are allowed (but not required) to enforce the expected ordering, and in practice some servers do so. Complete ordering requirements are given in Section 5.2 (The Order of Elements within the Deployment Descriptor), but note here that all `filter` elements must come before any `filter-mapping` elements, which in turn must come before any `servlet` or `servlet-mapping` elements.

### Core Warning

*Be sure to put all your `filter` and `filter-mapping` elements before any `servlet` and `servlet-mapping` elements in* web.xml.

For instance, given the `ReportFilter` class just shown, you could make the following `filter` declaration in *web.xml*. It associates the name `Reporter` with the actual class `ReportFilter` (which is in the `moreservlets` package).

```
<filter>
  <filter-name>Reporter</filter-name>
  <filter-class>moreservlets.ReportFilter</filter-class>
</filter>
```

Once you have named a filter, you associate it with one or more servlets or JSP pages by means of the `filter-mapping` element. You have two choices in this regard.

First, you can use `filter-name` and `servlet-name` subelements to associate the filter with a specific servlet name (which must be declared with a `servlet` element later in the same *web.xml* file). For example, the following snippet instructs the system to run the filter named `Reporter` whenever the servlet or JSP page named `SomeServletName` is accessed by means of a custom URL.

```
<filter-mapping>
  <filter-name>Reporter</filter-name>
  <servlet-name>SomeServletName</servlet-name>
</filter-mapping>
```

Second, you can use the `filter-name` and `url-pattern` subelements to associate the filter with groups of servlets, JSP pages, or static content. For example, the following snippet instructs the system to run the filter named `Reporter` when *any* URL in the Web application is accessed.

```
<filter-mapping>
  <filter-name>Reporter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

For example, Listing 5.12 shows a portion of a *web.xml* file that associates the ReportFilter filter with the servlet named PageName. The name PageName, in turn, is associated with a JSP file named *TestPage.jsp* and URLs that begin with the pattern *http://host/webAppPrefix/UrlTest2/*. The source code for *TestPage.jsp* and a discussion of the naming of JSP pages were given earlier in Section 5.3 (Assigning Names and Custom URLs). In fact, the servlet and servlet-name entries in Listing 5.12 are taken unchanged from that section. Given these *web.xml* entries, you see debugging reports in the standard output of the following sort (line breaks added for readability).

```
audit.irs.gov tried to access

http://mycompany.com/deployDemo/UrlTest2/business/tax-plan.html

on Tue Dec 25 13:12:29 EDT 2001.
```

**Listing 5.12**   *web.xml* (Excerpt showing filter usage)

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <filter>
    <filter-name>Reporter</filter-name>
    <filter-class>moreservlets.ReportFilter</filter-class>
  </filter>
  <!-- ... -->
  <filter-mapping>
    <filter-name>Reporter</filter-name>
    <servlet-name>PageName</servlet-name>
  </filter-mapping>
  <!-- ... -->
  <servlet>
    <servlet-name>PageName</servlet-name>
    <jsp-file>/TestPage.jsp</jsp-file>
  </servlet>
  <!-- ... -->
  <servlet-mapping>
    <servlet-name>PageName</servlet-name>
    <url-pattern>/UrlTest2/*</url-pattern>
  </servlet-mapping>
  <!-- ... -->
</web-app>
```

# 5.7 Specifying Welcome Pages

Suppose a user supplies a URL like *http://host/webAppPrefix/directoryName/* that contains a directory name but no filename. What happens? Does the user get a directory listing? An error? The contents of a standard file? If so, which one—*index.html*, *index.jsp*, *default.html*, *default.htm*, or what?

The `welcome-file-list` element, along with its subsidiary `welcome-file` element, resolves this ambiguity. For example, the following *web.xml* entry specifies that if a URL gives a directory name but no filename, the server should try *index.jsp* first and *index.html* second. If neither is found, the result is server specific (e.g., a directory listing).

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

Although many servers follow this behavior by default, they are not required to do so. As a result, it is good practice to explicitly use `welcome-file-list` to ensure portability.

**Core Approach**

*Make `welcome-file-list` a standard entry in your* web.xml *files.*

# 5.8 Designating Pages to Handle Errors

Now, I realize that *you* never make any mistakes when developing servlets and JSP pages and that all of your pages are so clear that no rational person could be confused by them. Still, however, the world is full of irrational people, and users could supply illegal parameters, use incorrect URLs, or fail to provide values for required form fields. Besides, *other* developers might not be as careful as you are, and they should have some tools to overcome their deficiencies.

The `error-page` element is used to handle problems. It has two possible subelements: `error-code` and `exception-type`. The first of these, `error-code`, designates what URL to use when a designated HTTP error code occurs. (If you aren't familiar with HTTP error codes, they are discussed in Chapter 6 of *Core Servlets and*

*JavaServer Pages*, which is available in PDF in its entirety at *http://www.more-servlets.com*.) The second of these subelements, `exception-type`, designates what URL to use when a designated Java exception is thrown but not caught. Both `error-code` and `exception-type` use the `location` element to designate the URL. This URL must begin with `/`. The page at the place designated by `location` can access information about the error by looking up two special-purpose attributes of the `HttpServletRequest` object: `javax.servlet.error.status_code` and `javax.servlet.error.message`.

Recall that it is important to declare the `web-app` subelements in the proper order within *web.xml*. Section 5.2 (The Order of Elements within the Deployment Descriptor) gives complete details on the required ordering. For now, however, just remember that `error-page` comes near the end of the *web.xml* file, after `servlet`, `servlet-name`, and `welcome-file-list`.

## The error-code Element

To better understand the value of the `error-code` element, consider what happens at most sites when you type the filename incorrectly. You typically get a 404 error message that tells you that the file can't be found but provides little useful information. On the other hand, try typing unknown filenames at *www.microsoft.com*, *www.ibm.com*, or especially *www.bea.com*. There, you get useful messages that provide alternative places to look for the page of interest. Providing such useful error pages is a valuable addition to your Web application. In fact, *http://www.plinko.net/404/* has an entire site devoted to the topic of 404 error pages. This site includes examples of the best, worst, and funniest 404 pages from around the world.

Listing 5.13 shows a JSP page that could be returned to clients that provide unknown filenames. Listing 5.14 shows the *web.xml* file that designates Listing 5.13 as the page that gets displayed when a 404 error code is returned. Figure 5–15 shows a typical result. Note that the URL displayed in the browser remains the one supplied by the client; the error page is a behind-the-scenes implementation technique.

Finally, remember that the default configuration of Internet Explorer version 5, in clear violation of the HTTP spec, ignores server-generated error messages and displays its own standard error message instead. Fix this by going to the Tools menu, selecting Internet Options, clicking on Advanced, then deselecting Show Friendly HTTP Error Messages.

### Core Warning

*In the default configuration, Internet Explorer improperly ignores server-generated error messages.*

**Listing 5.13**   *NotFound.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>404: Not Found</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2>Error!</H2>
I'm sorry, but I cannot find a page that matches
<%= request.getRequestURI() %> on the system. Maybe you should
try one of the following:
<UL>
  <LI>Go to the server's <A HREF="/">home page</A>.
  <LI>Search for relevant pages.<BR>
      <FORM ACTION="http://www.google.com/search">
      <CENTER>
      Keywords: <INPUT TYPE="TEXT" NAME="q"><BR>
      <INPUT TYPE="SUBMIT" VALUE="Search">
      </CENTER>
      </FORM>
  <LI>Admire a random multiple of 404:
      <%= 404*((int)(1000*Math.random())) %>.
  <LI>Try a <A HREF="http://www.plinko.net/404/rndindex.asp"
             TARGET="_blank">
      random 404 error message</A>. From the amazing and
      amusing plinko.net <A HREF="http://www.plinko.net/404/">
      404 archive</A>.
</UL>
</BODY></HTML>
```

**Listing 5.14**   *web.xml* (Excerpt designating error pages for HTTP error codes)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <error-page>
    <error-code>404</error-code>
    <location>/NotFound.jsp</location>
  </error-page>
  <!-- ... -->
</web-app>
```
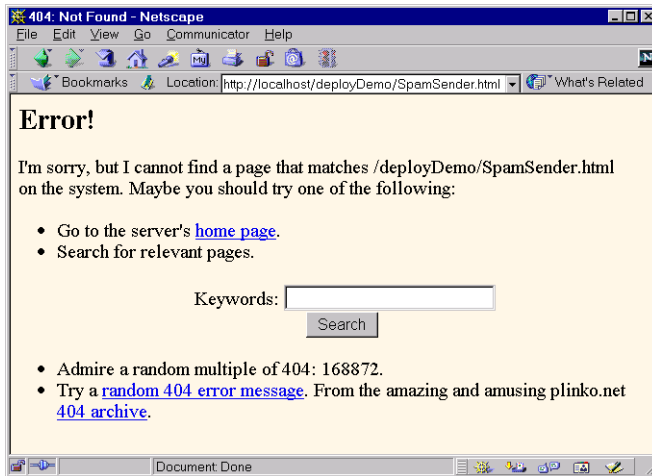
**Figure 5–15**   Use of helpful 404 messages can enhance the usability of your site.

## The exception-type Element

The `error-code` element handles the case when a request results in a particular HTTP status code. But what about the equally common case when the servlet or JSP page returns 200 but generates a runtime exception? That's the situation handled by the `exception-type` element. You only need to supply two things: a fully qualified exception class and a location, as below:

```
<error-page>
  <exception-type>packageName.className</exception-type>
  <location>/SomeURL</location>
</error-page>
```

Then, if any servlet or JSP page in the Web application generates an uncaught exception of the specified type, the designated URL is used. The exception type can be a standard one like `javax.ServletException` or `java.lang.OutOf-MemoryError`, or it can be an exception specific to your application.

For instance, Listing 5.15 shows an exception class named `DumbDeveloper-Exception` that might be used to flag particularly knuckle-headed mistakes by clueless programmers (not that you have any of those types on *your* development team). The class also contains a static method called `dangerousComputation` that sometimes generates this type of exception. Listing 5.16 shows a JSP page that calls `dangerousComputation` on random integer values. When the exception is thrown, *DDE.jsp* (Listing 5.17) is displayed to the client, as designated by the `exception-type` entry shown in the *web.xml* version of Listing 5.18. Figures 5–16 and 5–17 show lucky and unlucky results, respectively.

---

**Listing 5.15** *DumbDeveloperException.java*

```java
package moreservlets;

/** Exception used to flag particularly onerous
    programmer blunders. Used to illustrate the
    exception-type web.xml element.
 */

public class DumbDeveloperException extends Exception {
  public DumbDeveloperException() {
    super("Duh. What was I *thinking*?");
  }

  public static int dangerousComputation(int n)
      throws DumbDeveloperException {
    if (n < 5) {
      return(n + 10);
    } else {
      throw(new DumbDeveloperException());
    }
  }
}
```

---

**Listing 5.16** *RiskyPage.jsp*

```jsp
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Risky JSP Page</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2>Risky Calculations</H2>
<%@ page import="moreservlets.*" %>
<% int n = ((int)(10 * Math.random())); %>
<UL>
  <LI>n: <%= n %>
  <LI>dangerousComputation(n):
      <%= DumbDeveloperException.dangerousComputation(n) %>
</UL>
</BODY></HTML>
```

---

**Listing 5.17** *DDE.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Dumb</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2>Dumb Developer</H2>
We're brain dead. Consider using our competitors.
</BODY></HTML>
```

**Listing 5.18** *web.xml* (Excerpt designating error pages for exceptions)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->
  <servlet>...</servlet>
  <!-- ... -->
  <error-page>
    <exception-type>
      moreservlets.DumbDeveloperException
    </exception-type>
    <location>/DDE.jsp</location>
  </error-page>
  <!-- ... -->
</web-app>
```
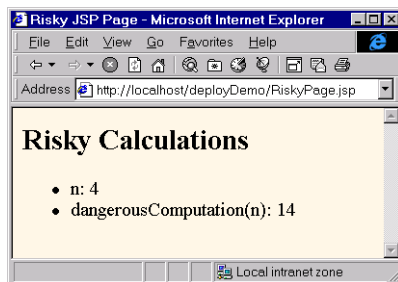
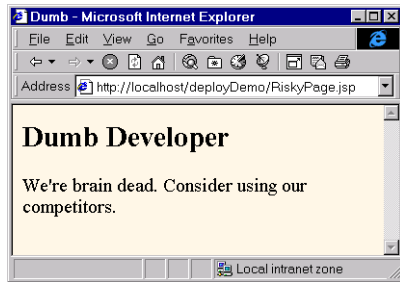**Figure 5–16** Fortuitous results of *RiskyPage.jsp*.

**Figure 5–17**   Unlucky results of *RiskyPage.jsp*.

# 5.9   Providing Security

Use of the server's built-in capabilities to manage security is discussed in Chapter 7 (Declarative Security). This section summarizes the *web.xml* elements that relate to this topic.

## Designating the Authorization Method

You use the `login-config` element to specify how the server should authorize users who attempt to access protected pages. It contains three possible subelements: `auth-method`, `realm-name`, and `form-login-config`. The `login-config` element should appear near the end of the *web.xml* deployment descriptor, immediately after the `security-constraint` element discussed in the next subsection. For complete details on the ordering of elements within *web.xml*, see Section 5.2. For details and examples on the use of the `login-config` element, see Chapter 7 (Declarative Security).

### auth-method
This subelement of `login-config` lists the specific authentication mechanism that the server should use. Legal values are `BASIC`, `DIGEST`, `FORM`, and `CLIENT-CERT`. Servers are only required to support `BASIC` and `FORM`.

`BASIC` specifies that standard HTTP authentication should be used, in which the server checks for an `Authorization` header, returning a 401 status code and a `WWW-Authenticate` header if the header is missing. This causes the client to pop up a dialog box that is used to populate the `Authorization` header. Details of this process are discussed in Section 7.3 (BASIC Authentication). Note that this mechanism provides little or no security against attackers

who are snooping on the Internet connection (e.g., by running a packet sniffer on the client's subnet) since the username and password are sent with the easily reversible base64 encoding. All compliant servers are required to support `BASIC` authentication.

`DIGEST` indicates that the client should transmit the username and password using the encrypted Digest Authentication form. This provides more security against network intercepts than does `BASIC` authentication, but the encryption can be reversed more easily than the method used in SSL (HTTPS). The point is somewhat moot, however, since few browsers currently support Digest Authentication, and consequently servlet containers are not required to support it.

`FORM` specifies that the server should check for a reserved session cookie and should redirect users who do not have it to a designated login page. That page should contain a normal HTML form to gather the username and password. After logging in, users are tracked by means of the reserved session-level cookie. Although in and of itself, `FORM` authentication is no more secure against network snooping than is `BASIC` authentication, additional protection such as SSL or network-level security (e.g., IPSEC or VPN) can be layered on top if necessary. All compliant servers are required to support `FORM` authentication.

`CLIENT-CERT` stipulates that the server must use HTTPS (HTTP over SSL) and authenticate users by means of their Public Key Certificate. This provides strong security against network intercept, but only J2EE-compliant servers are required to support it.

### realm-name
This element applies only when the `auth-method` is `BASIC`. It designates the name of the security realm that is used by the browser in the title of the dialog box and as part of the `Authorization` header.

### form-login-config
This element applies only when the `auth-method` is `FORM`. It designates two pages: the page that contains the HTML form that collects the username and password (by means of the `form-login-page` subelement), and the page that should be used to indicate failed authentication (by means of the `form-error-page` subelement). As discussed in Chapter 7, the HTML form given by the `form-login-page` must have an `ACTION` attribute of `j_security_check`, a username textfield named `j_username`, and a password field named `j_password`.

For example, Listing 5.19 instructs the server to use form-based authentication. A page named *login.jsp* in the top-level directory of the Web app should collect the username and password, and failed login attempts should be reported by a page named *login-error.jsp* in the same directory.

**Listing 5.19**   *web.xml* (Excerpt showing `login-config`)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->
  <security-constraint>...</security-constraint>
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/login.jsp</form-login-page>
      <form-error-page>/login-error.jsp</form-error-page>
    </form-login-config>
  </login-config>
  <!-- ... -->
</web-app>
```

## Restricting Access to Web Resources

So, you can tell the server which authentication method to use. "Big deal," you say, "that's not much use unless I can designate the URLs that ought to be protected." Right. Designating these URLs and describing the protection they should have is the purpose of the `security-constraint` element. This element should come immediately *before* `login-config` in *web.xml.* It contains four possible subelements: `web-resource-collection`, `auth-constraint`, `user-data-constraint`, and `display-name`. Each of these is described in the following subsections.

### web-resource-collection

This element identifies the resources that should be protected. All `security-constraint` elements must contain at least one `web-resource-collection` entry. This element consists of a `web-resource-name` element that gives an arbitrary identifying name, a `url-pattern` element that identifies the URLs that should be protected, an optional `http-method` element that designates the HTTP commands to which the protection applies

(GET, POST, etc.; the default is all methods), and an optional `description` element that provides documentation. For example, the following `web-resource-collection` entry (within a `security-constraint` element) designates that all documents in the *proprietary* directory of the Web application should be protected.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Proprietary</web-resource-name>
    <url-pattern>/proprietary/*</url-pattern>
  </web-resource-collection>
  <!-- ... -->
</security-constraint>
```

It is important to note that the `url-pattern` applies only to clients that access the resources directly. In particular, it does *not* apply to pages that are accessed through the MVC architecture with a `RequestDispatcher` or by the similar means of `jsp:forward`. This asymmetry is good if used properly. For example, with the MVC architecture a servlet looks up data, places it in beans, and forwards the request to a JSP page that extracts the data from the beans and displays it (see Section 3.8). You want to ensure that the JSP page is never accessed directly but instead is accessed only through the servlet that sets up the beans the page will use. The `url-pattern` and `auth-constraint` (see next subsection) elements can provide this guarantee by declaring that *no* user is permitted direct access to the JSP page. But, this asymmetric behavior can catch developers off guard and allow them to accidentally provide unrestricted access to resources that should be protected.

### Core Warning

*These protections apply only to direct client access. The security model does not apply to pages accessed by means of a `RequestDispatcher` or `jsp:forward`.*

#### auth-constraint

Whereas the `web-resource-collection` element designates which URLs should be protected, the `auth-constraint` element designates which users should have access to protected resources. It should contain one or more `role-name` elements identifying the class of users that have access and, optionally, a `description` element describing the role. For instance, the following part of the `security-constraint` element in *web.xml* states that

only users who are designated as either Administrators or Big Kahunas (or both) should have access to the designated resource.

```
<security-constraint>
  <web-resource-collection>...</web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
    <role-name>kahuna</role-name>
  </auth-constraint>
</security-constraint>
```

It is important to realize that this is the point at which the portable portion of the process ends. How a server determines which users are in which roles and how it stores user passwords is completely system dependent. See Section 7.1 (Form-Based Authentication) for information on the approaches used by Tomcat, JRun, and ServletExec.

For example, Tomcat uses *install_dir/conf/tomcat-users.xml* to associate user-names with role names and passwords, as in the example below that designates users joe (with password bigshot) and jane (with password enaj) as belonging to the administrator and/or kahuna roles.

```
<tomcat-users>
  <user name="joe"
        password="bigshot" roles="administrator,kahuna" />
  <user name="jane"
        password="enaj" roles="kahuna" />
  <!-- ... -->
</tomcat-users>
```

### Core Warning

*Container-managed security requires a significant server-specific component. In particular, you must use nonportable methods to associate passwords with usernames and to map usernames to role names.*

**user-data-constraint**
This optional element indicates which transport-level protections should be used when the associated resource is accessed. It must contain a transport-guarantee subelement (with legal values NONE, INTEGRAL, or CONFIDEN-TIAL) and may optionally contain a description element. A value of NONE (the default) for transport-guarantee puts no restrictions on the commu-

nication protocol used. A value of INTEGRAL means that the communication must be of a variety that prevents data from being changed in transit without detection. A value of CONFIDENTIAL means that the data must be transmitted in a way that prevents anyone who intercepts it from reading it. Although in principle (and in future HTTP versions) there may be a distinction between INTEGRAL and CONFIDENTIAL, in current practice they both simply mandate the use of SSL. For example, the following instructs the server to only permit HTTPS connections to the associated resource:

```
<security-constraint>
  <!-- ... -->
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

**display-name**
This rarely used subelement of security-constraint gives a name to the security constraint entry that might be used by a GUI tool.

## Assigning Role Names

Up to this point, the discussion has focused on security that was completely managed by the container (server). Servlets and JSP pages, however, can also manage their own security. For details, see Chapter 8 (Programmatic Security).

For example, the container might let users from either the bigwig or big-cheese role access a page showing executive perks but permit only the bigwig users to modify the page's parameters. One common way to accomplish this more fine-grained control is to call the isUserInRole method of HttpServlet-Request and modify access accordingly (for an example, see Section 8.2).

The security-role-ref subelement of servlet provides an alias for a security role name that appears in the server-specific password file. For instance, suppose a servlet was written to call request.isUserInRole("boss") but is then used in a server whose password file calls the role manager instead of boss. The following would permit the servlet to use either name.

```
<servlet>
  <!-- ... -->
  <security-role-ref>
    <role-name>boss</role-name>    <!-- New alias -->
    <role-link>manager</role-link> <!-- Real name -->
  </security-role-ref>
</servlet>
```

You can also use a `security-role` element within `web-app` to provide a global list of all security roles that will appear in the `role-name` elements. Declaring the roles separately could make it easier for advanced IDEs to manipulate security information.

# 5.10  Controlling Session Timeouts

If a session has not been accessed for a certain period of time, the server can throw it away to save memory. You can explicitly set the timeout for individual session objects by using the `setMaxInactiveInterval` method of `HttpSession`. If you do not use this method, the default timeout is server specific. However, the `session-config` and `session-timeout` elements can be used to give an explicit timeout that will apply on all servers. The units are minutes, so the following example sets the default session timeout to three hours (180 minutes).

```
<session-config>
  <session-timeout>180</session-timeout>
</session-config>
```

# 5.11  Documenting Web Applications

More and more development environments are starting to provide explicit support for servlets and JSP. Examples include Borland JBuilder Enterprise Edition, Macromedia UltraDev, Allaire JRun Studio, and IBM VisualAge for Java.

A number of the *web.xml* elements are designed not for the server, but for the visual development environment. These include `icon`, `display-name`, and `description`.

Recall that it is important to declare the `web-app` subelements in the proper order within *web.xml*. Section 5.2 (The Order of Elements within the Deployment Descriptor) gives complete details on the required ordering. For now, however, just remember that `icon`, `display-name`, and `description` are the first three legal elements within the `web-app` element of *web.xml*.

**icon**
The `icon` element designates the location of either one or two image files that the GUI tool can use to represent the Web application. A 16 × 16 GIF or JPEG image can be specified with the `small-icon` element, and a 32 × 32 image can be specified with `large-icon`. Here is an example:

```
<icon>
   <small-icon>/images/small-book.gif</small-icon>
   <large-icon>/images/tome.jpg</large-icon>
</icon>
```

### display-name

The `display-name` element provides a name that the GUI tools might use to label this particular Web application. Here is an example.

```
<display-name>Rare Books</display-name>
```

### description

The `description` element provides explanatory text, as below.

```
<description>
This Web application represents the store developed for
rare-books.com, an online bookstore specializing in rare
and limited-edition books.
</description>
```

# 5.12 Associating Files with MIME Types

Servers typically have a way for Webmasters to associate file extensions with media types. So, for example, a file named *mom.jpg* would automatically be given a MIME type of `image/jpeg`. However, suppose that your Web application has unusual files that you want to guarantee are assigned a certain MIME type when sent to clients. The `mime-mapping` element, with `extension` and `mime-type` subelements, can provide this guarantee. For example, the following code instructs the server to assign a MIME type of `application/x-fubar` to all files that end in *.foo*.

```
<mime-mapping>
   <extension>foo</extension>
   <mime-type>application/x-fubar</mime-type>
</mime-mapping>
```

Or, perhaps your Web application wants to override standard mappings. For instance, the following would tell the server to designate *.ps* files as plain text (`text/ plain`) rather than as PostScript (`application/postscript`) when sending them to clients.

```
<mime-mapping>
  <extension>ps</extension>
  <mime-type>text/plain</mime-type>
</mime-mapping>
```

For more information on MIME types, see Table 2.1 on page 88.

# 5.13  Locating Tag Library Descriptors

The JSP `taglib` element has a required `uri` attribute that gives the location of a Tag Library Descriptor (TLD) file relative to the Web application root. The actual name of the TLD file might change when a new version of the tag library is released, but you might want to avoid changing all the existing JSP pages. Furthermore, you might want to use a short `uri` to keep the `taglib` elements concise. That's where the deployment descriptor's `taglib` element comes in. It contains two subelements: `taglib-uri` and `taglib-location`. The `taglib-uri` element should exactly match whatever is used for the `uri` attribute of the JSP `taglib` element. The `taglib-location` element gives the real location of the TLD file. For example, suppose that you place the file *chart-tags-1.3beta.tld* in *yourWebApp/WEB-INF/tlds*. Now, suppose that *web.xml* contains the following within the `web-app` element.

```
<taglib>
  <taglib-uri>/charts.tld</taglib-uri>
  <taglib-location>
    /WEB-INF/tlds/chart-tags-1.3beta.tld
  </taglib-location>
</taglib>
```

Given this specification, JSP pages can now make use of the tag library by means of the following simplified form.

```
<% taglib uri="/charts.tld" prefix="somePrefix" %>
```

# 5.14  Designating Application Event Listeners

Application event listeners are classes that are notified when the servlet context or a session object is created or modified. They are new in version 2.3 of the servlet specification and are discussed in detail in Chapter 10 (The Application Events Frame-

work). Here, though, I just want to briefly illustrate the use of the *web.xml* elements that are used to register a listener with the Web application.

Registering a listener involves simply placing a `listener` element inside the `web-app` element of *web.xml*. Inside the `listener` element, a `listener-class` element lists the fully qualified class name of the listener, as below.

```
<listener>
  <listener-class>package.ListenerClass</listener-class>
</listener>
```

Although the structure of the `listener` element is simple, don't forget that you have to properly order the subelements inside the `web-app` element. The `listener` element goes immediately before all the `servlet` elements and immediately after any `filter-mapping` elements. Furthermore, since application life-cycle listeners are new in version 2.3 of the servlet specification, you have to use the 2.3 version of the *web.xml* DTD, not the 2.2 version.

For example, Listing 5.20 shows a simple listener called `ContextReporter` that prints a message on the standard output whenever the Web application's `Servlet-Context` is created (e.g., the Web application is loaded) or destroyed (e.g., the server is shut down). Listing 5.21 shows the portion of the *web.xml* file that is required for registration of the listener.

---

**Listing 5.20**    *ContextReporter.java*

```java
package moreservlets;

import javax.servlet.*;
import java.util.*;

/** Simple listener that prints a report on the standard output
 *  when the ServletContext is created or destroyed.
 */

public class ContextReporter implements ServletContextListener {
  public void contextInitialized(ServletContextEvent event) {
    System.out.println("Context created on " +
                       new Date() + ".");
  }

  public void contextDestroyed(ServletContextEvent event) {
    System.out.println("Context destroyed on " +
                       new Date() + ".");
  }
}
```

---

| **Listing 5.21** | *web.xml* (Excerpt declaring a listener) |
|---|---|

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <!-- ... -->
  <filter-mapping>...</filter-mapping>
  <listener>
    <listener-class>moreservlets.ContextReporter</listener-class>
  </listener>
  <servlet>...</servlet>
  <!-- ... -->
</web-app>
```

## 5.15 J2EE Elements

This section describes the *web.xml* elements that are used for Web applications that are part of a J2EE environment. I'll provide a brief summary here; for details, see Chapter 5 of the Java 2 Platform Enterprise Edition version 1.3 specification at *http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf*.

### distributable

The `distributable` element indicates that the Web application is programmed in such a way that servers that support clustering can safely distribute the Web application across multiple servers. For example, a distributable application must use only `Serializable` objects as attributes of its `HttpSession` objects and must avoid the use of instance variables (fields) for implementing persistence. The `distributable` element appears directly after the `description` element (Section 5.11) and contains no subelements or data—it is simply a flag (as below).

```
<distributable />
```

### resource-env-ref

The `resource-env-ref` element declares an administered object associated with a resource. It consists of an optional `description` element, a `resource-env-ref-name` element (a JNDI name relative to the

`java:comp/env` context), and a `resource-env-type` element (the fully qualified class designating the type of the resource), as below.

```
<resource-env-ref>
  <resource-env-ref-name>
    jms/StockQueue
  </resource-env-ref-name>
  <resource-env-ref-type>
    javax.jms.Queue
  </resource-env-ref-type>
</resource-env-ref>
```

### resource-ref

The `resource-ref` element declares an external resource used with a resource factory. It consists of an optional `description` element, a `res-ref-name` element (the resource manager connection-factory reference name), a `res-type` element (the fully qualified class name of the factory type), a `res-auth` element (the type of authentication used—`Application` or `Container`), and an optional `res-sharing-scope` element (a specification of the shareability of connections obtained from the resource—`Shareable` or `Unshareable`). Here is an example.

```
<resource-ref>
  <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope
</resource-ref>
```

### env-entry

The `env-entry` element declares the Web application's environment entry. It consists of an optional `description` element, an `env-entry-name` element (a JNDI name relative to the `java:comp/env` context), an `env-entry-value` element (the entry value), and an `env-entry-type` element (the fully qualified class name of a type in the `java.lang` package— `java.lang.Boolean`, `java.lang.String`, etc.). Here is an example.

```
<env-entry>
  <env-entry-name>minAmount</env-entry-name>
  <env-entry-value>100.00</env-entry-value>
  <env-entry-type>java.lang.Double</env-entry-type>
</env-entry>
```

### ejb-ref

The `ejb-ref` element declares a reference to the home of an enterprise bean. It consists of an optional `description` element, an `ejb-ref-name` element (the name of the EJB reference relative to `java:comp/env`), an `ejb-ref-type` element (the type of the bean—`Entity` or `Session`), a `home` element (the fully qualified name of the bean's home interface), a `remote` element (the fully qualified name of the bean's remote interface), and an optional `ejb-link` element (the name of another bean to which the current bean is linked).

### ejb-local-ref

The `ejb-local-ref` element declares a reference to the local home of an enterprise bean. It has the same attributes and is used in the same way as the `ejb-ref` element, with the exception that `local-home` is used in place of `home`.

**J2EE training from the author!** Marty Hall, author of five bestselling books from Prentice Hall (including this one), is available for customized J2EE training. Distinctive features of his courses:

- Marty developed all his own course materials: no materials licensed from some unknown organization in Upper Mongolia.
- Marty personally teaches all of his courses:
  no inexperienced flunky regurgitating memorized PowerPoint slides.
- Marty has taught thousands of developers in the USA, Canada, Australia, Japan, Puerto Rico, and the Philippines: no first-time instructor using your developers as guinea pigs.
- Courses are available onsite at *your* organization (US and internationally): cheaper, more convenient, and more flexible. Customizable content!
- Courses are also available at public venues:
  for organizations without enough developers for onsite courses.
- Many topics are available: intermediate servlets & JSP, advanced servlets & JSP, Struts, JSF, Java 5, AJAX, and more. Custom combinations of topics are available for onsite courses.

Need more details? Want to look at sample course materials? Check out http://courses.coreservlets.com/. Want to talk directly to the instructor about a possible course? Email Marty at hall@coreservlets.com.